UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1298

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

LES PRÉDICATS COLLECTIFS : UN MOYEN D'EXPRESSION DU CONTRÔLE DU PARALLÉLISME *OU* EN PROLOG

René QUINIOU
Laurent TRILLING

Octobre 1990

RR-1298

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTEMES ALEATOIRES

Les Prédicats Collectifs : un moyen d'expression du contrôle du parallélisme OU en Prolog

Collective Predicates: Expressing Control of OR-Parallelism in Prolog

René QUINIOU - Laurent TRILLING*
IRISA - INRIA
Campus de Beaulieu
35042 RENNES Cedex
FRANCE

Publication Interne n° 548 - Septembre 1990, 34 Pages

Résumé : La plupart des modèles de parallélisme proposés pour la programmation en logique ne possèdent pas d'outils permettant d'exprimer un contrôle entre des évaluations concurrentes. Tel est l'objectif des prédicats collectifs qui se présentent comme une extension du prédicat d'ordre supérieur `bagof(X,LVG,F,LX)` satisfait si `LX` est la liste des instances de `X` telles que `F` soit satisfait (pour des valeurs données des variables appartenant à la liste des variables `LVG`). L'évaluation de ces prédicats utilise des informations provenant de résolvantes concurrentes. Ils permettent ainsi de spécifier clairement des méthodes de contrôle sophistiquées pour la recherche heuristique de solutions. Nous donnons ici, à titre d'illustration, plusieurs expressions de l'algorithme A*, en améliorant pas-à-pas la version initiale de l'algorithme. Nous présentons, enfin, des extensions possibles et les caractéristiques de l'implantation.

Mots-clés : parallélisme, synchronisation, Prolog, algorithme de contrôle A*

Abstract : Most of the parallel models proposed for logic programming lack tools for expressing control between concurrent evaluations. This is the goal of collective predicates which look like an extension of the higher order predicate `bagof(X,LVG,F,LX)` satisfied if `LX` is the list of instances of `X` such that `F` can be proved (for given value to variables belonging to the list `LVG`). The evaluation of such predicates uses informations coming from concurrent resolvents. So, they allow the clear specification of sophisticated control methods for heuristic search of solutions. Here, we illustrate this point on several expressions of the A* algorithm, improving step by step the initial version of the algorithm. Finally, we present possible extensions and the main implementation features.

Key-words : logic programming, parallelism, synchonisation, Prolog, A* algorithm

* Adresse actuelle: IMAG-LGI Bât D 38402 St Martin d'Hères Cedex

1 Introduction

Le problème que nous abordons est le suivant : dans la mesure où les algorithmes dits de contrôle sont après tout fondés logiquement, comment peut-on les exprimer grâce à une extension de Prolog.

Beaucoup de propositions permettant le réordonnancement des sous-buts lors de l'exécution ont été proposées. Le prédicat `freeze` de Prolog II [Colmérauer 83] ou les primitives d'Epilog [Porto 82] constituent des exemples de telles propositions. Beaucoup moins de propositions concernent le contrôle entre résolvantes ayant pour origine le même sous-but [Nakamura 86, Quiniou 85]. Nakamura propose d'ajouter une nouvelle primitive appelée `hold(V)` à Prolog. Celle-ci permet la suspension de l'évaluation des résolvantes. Seules les résolvantes correspondant à la valeur de V minimale sont ensuite relancées. Les prédicats "collectifs", que l'on présente ici, constituent un moyen de résoudre des problèmes plus généraux que ceux permis par la primitive de Nakamura : l'examen global des résolvantes est réalisé grâce à l'utilisation de prédicats définis en Prolog par le programmeur lui-même ce qui est à distinguer d'une fonction définie une fois pour toutes (telle que la fonction minimum pour Nakamura).

Il faut noter que tous les systèmes Prolog possèdent le moyen d'exprimer un contrôle rudimentaire entre résolvantes par le biais du prédicat prédéfini `bagof(X,LVG,F,LX)` satisfait si LX est la liste (multi-ensemble) des instances de X telles que F soit démontrable (pour des valeurs données des variables appartenant à la liste des variables LVG). Notre idée consiste à introduire une extension de ce prédicat, soit `sync_list_of(X,LVG,F,LX,Q)`, où Q définit un prédicat "collectif" qui permet la synchronisation des résolvantes engendrées par F. Cette synchronisation est assurée par des littéraux de la forme `of_list(PC)` où PC est un littéral de prédicat dit "collectif". L'évaluation d'un littéral `of_list(PC)` provoque la suspension de la résolvante où il apparaît jusqu'à la fin de l'évaluation du prédicat collectif.

Nous introduisons tout d'abord ces notions sur un exemple puis nous en donnons une description plus formelle. La suite de l'article est consacré à l'illustration des capacités des prédicats collectifs à exprimer l'algorithme de contrôle A* [Nilsson 79]. Nous débutons par l'expression du problème à résoudre, c'est-à-dire la détermination des solutions accessibles par le plus court chemin, puis nous en dérivons une expression avec un contrôle de type A* d'abord avec synchronisation globale ensuite avec synchronisations locales. De cette façon, la partie contrôle que nous introduisons se distingue clairement du problème lui-même. Nous concluons sur les cas particuliers, les extensions possibles et l'implantation de ces prédicats collectifs. On trouve en annexe la définition en termes logiques de ces prédicats ainsi que leur mise en œuvre décrite sous la forme d'un interpréteur PROLOG.

2 Introduction informelle des prédicats collectifs

Partant d'un problème simple, nous essayons tout d'abord de le formuler en mettant en relief ses possibilités de "répartition" grâce à l'utilisation du prédicat `bagof`. Nous soulignons les inconvénients de cette approche et introduisons la notion de prédicats collectifs. Après une description informelle de leur signification, nous donnons une définition plus précise en nous appuyant toujours sur le même exemple. Nous terminons ce chapitre par la présentation d'un exemple plus complexe, montrant comment un prédicat collectif peut être utilisé dans la définition d'un autre prédicat collectif.

2.1 Formulation initiale

Considérons le problème simple suivant :

`pierre`, `jean` et `joseph` mangent respectivement `10`, `18` et `32` kg de nourriture, les premiers de la viande et le dernier des légumes. Ceci est modélisé par les clauses suivantes :

```
mange(pierre,viande,10).
mange(jean,viande,18).
mange(joseph,légumes,32).
```

Supposons, de plus, que `pierre` et `jean` soient omnivores et `joseph` végétarien.

Définissons un mangeur raisonnable comme un omnivore qui mange moins qu'une certaine limite et fixons cette limite à la moyenne des quantités de nourriture ingérées par l'ensemble des personnes. Il s'agit d'établir la liste de tous les mangeurs raisonnables.

Une première idée consiste à utiliser le prédicat prédéfini `bagof(X,LVG,F,LX)` satisfait si `LX` est la collection (avec, éventuellement, des répétitions) des instances de `X` satisfaisant le but `F` et obtenues pour une même valeur de `V`, pour toute variable `V` appartenant à la liste `LVG` [Warren 82], [Naish 84]. Ces variables sont appelées variables globales. Il faut noter, donc, que le prédicat `bagof` est non déterministe : à chaque instanciation possible des variables de `LVG` peut lui correspondre une solution. Le prédicat `mangeurs_raisonnables(LX)`, satisfait si `LX` est la liste de tous les mangeurs raisonnables, peut alors se définir comme suit :

```
mangeurs_raisonnables(LX) :-
        bagof([X,Q],[],mange(X,B,Q),LXQ),
        moyenne_lxq(LXQ,M),
        bagof(X,[M,LXQ],(XQ=[X,Q],XQ ∈ LXQ,Q<M,omnivore(X)),LX).
```

où `moyenne_lxq(LXQ,M)` est satisfait si `M` est la moyenne des quantités de nourriture contenues dans la liste `LXQ` (composée de couples individus, quantité de nourriture). Si l'on admet que `joseph` n'est pas omnivore, `mangeurs_raisonnables(LX)` est satisfait pour `LX=[pierre,jean]` avec `LXQ=[[pierre,10],[jean,18],[joseph,32]]` et `M = 60/3 = 20`.

Cette expression du prédicat `mangeurs_raisonnables` nous semble intéressante dans la mesure où elle fait apparaître l'indépendance de certaines évaluations. On peut considérer, en effet, que l'évaluation d'un littéral de la forme `bagof(X,LVG,F,LX)` mène à toutes les évaluations indépendantes de `F` (partageant les variables globales contenues dans `LVG`). Pratiquement, cela signifie qu'un certain degré de parallélisme peut être atteint. Les prédicats collectifs, qui sont introduits dans la suite, permettent une expression plus fine des évaluations indépendantes et potentiellement donc une efficacité plus grande à l'exécution. Intuitivement, sur notre exemple, il s'agit d'éviter le partage en deux évaluations "réparties" communiquant par une évaluation "centralisée" (`moyenne_lxq(LXQ,M)`) qui utilise une structure trop complexe pour son objet (la liste `LXQ` n'est pas nécessaire pour déterminer la moyenne `M`, il suffirait d'une liste des quantités, c'est-à-dire dans notre exemple `LQ=[10,18,32]`). L'objectif consiste, plutôt, à exprimer des évaluations indépendantes telles que "chaque mangeur décide lui-même s'il est raisonnable ou non" et communiquant entre elles pour la détermination de `M` en n'échangeant que les informations strictement nécessaires. Et, bien sûr, on impose de ne faire qu'une seule évaluation du littéral `mange(X,B,Q)` par processus.

2.2 Introduction des prédicats collectifs

Le prédicat `mangeurs_raisonnables` peut s'écrire de la manière suivante au moyen du prédicat collectif `limite`

```
mangeurs_raisonnables(LX) :-
        sync_list_of(X,[],(mangeurs(X,B,Q),of_list(limite(Q),omnivore(X))),LX,
                        [Q,limite(Q),LQ,[moyenne(LQ,M),Q<M]]).
```

Intuitivement et opérationnellement, l'évaluation du but `mangeurs_raisonnables(LX)` peut être vue comme l'activation de trois processus indépendants (un pour chacun des individus faisant partie de la communauté des mangeurs : `pierre`, `jean`, `joseph`) évaluant chacun `mange(X,B,Q)` puis se synchronisant pour procéder à l'évaluation de leurs littéraux `of_list(limite(Q),omnivore(X))`. Les premiers paramètres de ces littéraux `of_list` (ici `limite(10)`, `limite(18)`, `limite(32)`) sont dits littéraux du prédicat collectif `limite`. Une variable `LQ`, globale à tous les processus, est alors créée. Sa valeur est la liste des premiers paramètres des littéraux du prédicat `of_list` (ici `LQ = [10,18,32]`). Ensuite `moyenne(LQ,M)` est évalué (ici `M = 20`). Enfin chacun des processus reprend son cours indépendant en évaluant `Q<M` (où `M` est globale, c'est-à-dire unique pour l'ensemble des processus) et enfin `omnivore(X)`. Un de ces processus mène à un échec (celui relatif à `joseph`) et la liste `LX` des `X` solutions est bien `[pierre, jean]`.

Plus logiquement, la satisfaction du but `mangeurs_raisonnables(LX)` peut être établie de la façon suivante :

- `LX` est la liste des `X` vérifiant le but `(mange(X,B,Q),list_of(limite(Q),omnivore(X)))`.

La seule différence entre le prédicat `bagof(X,LVG,F,LX)` et le prédicat `sync_list_of(X,LVG,F',LX,[E,PC,LE,[G,L]])` réside dans le fait que le but `F'` peut être défini à l'aide de littéraux de la forme `of_list(PC',R)` où `PC'` est un littéral du prédicat collectif défini par le cinquième argument et `R` une suite de buts.

• Le but visé, c'est-à-dire `(mange(X,B,Q),of_list(limite(Q),omnivore(X)))`, ne peut être satisfait que si :

(1)
```
of_list (limite(10),omnivore(pierre)) ∨
of_list (limite(18),omnivore(jean)) ∨
of_list (limite(32),omnivore(joseph))
```

est satisfait.

Pour prouver (1), nous appliquons une nouvelle règle d'inférence permettant de conclure que (1) n'est satisfait que si

(2)
```
(moyenne([10,18,32],M),
   ( (10<M,omnivore(pierre)) ∨
     (18<M,omnivore(jean)) ∨
     (32<M,omnivore(joseph))  )  )
```

est satisfait. L'expression (2) est construite de la façon suivante :

a) On considère l'ensemble `EC` des littéraux de prédicats collectifs apparaissant dans (1) (dans notre exemple `EC = {limite(10), limite(18), limite(32)}`) et le cinquième paramètre du littéral `sync_list_of` à évaluer (ici, `[Q,limite(Q),LQ,[moyenne(LQ,M),Q<M]]`) : on introduit la liste `LQ` des instances de `Q` résultant de l'unification de `limite(Q)` avec chacun des `C ∈ EC` (ici `LQ=[10,18,32]`).

b) On déduit la formule (2) de (1) en remplaçant chaque littéral `of_list(limite(Q),omnivore(X))` de (1) par `(moyenne(LQ,M),Q<M,omnivore(X))` et en "factorisant" `moyenne(LQ,M)` : cela revient à considérer que `LQ` et `M` sont des variables globales.

Nous obtenons donc par cette formulation une définition de `mangeurs_raisonnables(LX)`, d'une part équivalente à celle donnée en 2.1 (la liste `LX` des individus satisfaisant (2) est bien `[pierre, jean]`) et exprimant, d'autre part, que les seules variables "à partager" sont `LQ` et `M` et non `LQX` comme c'était le cas au paragraphe précédent.

2.3 Définition plus précise des prédicats collectifs

Pour ce faire, il faut revenir sur la définition de la nouvelle règle d'inférence introduite en précisant d'abord la structure du cinquième argument du prédicat `sync_list_of(X,LVG,F',LX,[E,PC,LE,[G,L]])` :

- E est un terme
- PC est un littéral de prédicat dit collectif
- LE est une liste de termes
- G et L sont des suites de buts.

Le cas à considérer est celui où F' ne peut être satisfaite que si une formule du type

(I) `of_list(PC1',R1) ∨ ... ∨ of_list(PCn',Rn)` avec $n \geq 1$

est satisfaite (c'est-à-dire que chacune des évaluations indépendantes de F' mène à l'évaluation d'un littéral `of_list(PCi',Ri)`, pour `i:1,n`) : on dit que ces littéraux forment une "collectivité". La nouvelle règle d'inférence dit essentiellement que (I) ne peut être satisfaite que si (II) telle que

(II) `G', ((L1',R1') ∨ ... ∨ (Ln',Rn'))`

est satisfaite avec

- G' est une variante de G dans laquelle LE est la liste des instances de E résultant des unifications de PC et de PCi' pour `i:1,n`.

- Li' (resp. Ri') pour `i:1,n` est obtenu en appliquant à Li (resp. Ri) la substitution permettant d'unifier PC et PCi' si elle existe : si cette substitution n'existe pas (PC et PCi' ne sont pas unifiables), on considère que Li' est faux.

Nous donnons en Annexe 1 une définition du prédicat `sync_list_of` en fonction du prédicat `bagof` et en Annexe 2 les règles permettant l'évaluation d'un littéral de prédicat `bagof` et une extension de ces règles permettant l'évaluation d'un littéral de prédicat `sync_list_of`.

2.4 Restriction d'emploi en programmation logique

En se situant clairement dans une optique d'exécution de Prolog, admettons, dans la suite, que toute liste de buts de la forme

```
A, of_list(PC',R), R'
```

doit être considérée comme signifiant :

```
A, of_list(PC', (R, R'))
```

Dans ces conditions, tout littéral `of_list(PC',R)` peut se ramener à une forme possédant une liste de buts à droite vide et il n'est pas nécessaire de donner un deuxième paramètre au prédicat `of_list`. Ceci permet d'alléger quelque peu l'écriture, ainsi que le montre la nouvelle formulation du prédicat `mangeurs_raisonnables` :

```
mangeurs_raisonnables(LX) :-
        sync_list_of(X,[],mangeur_raisonnable(X),LX,
                     [Q,limite(Q),LQ,[moyenne(LQ,M),Q<M]]).

mangeur_raisonnable(X) :-
        mange(X,B,Q),
        of_list(limite(Q)),
        omnivore(X).
```

Il faut cependant être bien conscient que, sous cette formulation, la deuxième règle n'est pas alors équivalente à la règle suivante :

```
mangeur_raisonnable(X) :-
        mange(X,B,Q),
        omnivore(X),
        of_list(limite(Q)).
```

Avec cette dernière règle, seuls les omnivores sont concernés par la synchronisation et donc `LQ = [10,18], M = (10+18)/2 = 14, LX = [pierre].`

On trouve en annexe une définition complète.

2.4 Exemple de prédicat collectif intervenant dans la définition d'un prédicat collectif

Considérons maintenant un problème plus complexe où un prédicat collectif est défini en fonction d'un autre prédicat collectif. Afin d'illustrer ceci, on reprend l'exemple précédent en tenant compte du fait que le viande contient environ quatre fois plus de protéines que les légumes. La limite à ne pas dépasser pour être raisonnable devient maintenant la moyenne pondérée des moyennes des consommations de viande et de légumes.

On introduit d'abord le prédicat `mangeurs_r(LX)` satisfait si tout `X` appartenant à `LX` est raisonnable et mange la même nourriture. On a alors :

```
mangeurs_raisonnables(L) :-
        sync_list_of(LX,[],mangeurs_r(LX),LLX,Dlimite),
        lineariser(LLX,L).
```

où `lineariser(LLX,L)` est satisfait si `L` est la liste des éléments apparaissant dans la liste de listes `LLX` et `Dlimite` est la définition du prédicat collectif `limite` dont nous examinons l'emploi ci-dessous.

Deux règles définissent `mangeurs_r(LX)`, l'une relative aux mangeurs de viande l'autre aux mangeurs de légumes.

```
mangeurs_r(LX) :-
        sync_list_of(X,[],mangeur_r_viande(X),LX,Dlimite_viande).
mangeurs_r(LX) :-
        sync_list_of(X,[],mangeur_r_légumes(X),LX,Dlimite_légumes).
```

où `mangeur_r_viande(X)` (resp. `mangeur_r_légumes(X)`) est satisfait si `X` est un mangeur

raisonnable de viande (resp. légumes) et où `Dlimite_viande` (resp. `Dlimite_légumes`) est la définition du prédicat collectif `limite_viande` (resp. `limite_légumes`) que nous précisons ci-dessous.

Les prédicats `mange_r_viande` et `mange_r_légumes` sont définis par :

```
mange_r_viande(X) :-
        mange(X,viande,Q), of_list(limite_viande(Q)), omnivore(X).
mange_r_légumes(X) :-
        mange(X,légumes,Q), of_list(limite_légumes(Q)), omnivore(X).
```

où `of_list(limite_viande(Q))` (resp. `of_list(limite_légumes(Q))`) est satisfait si `Q` est une quantité inférieure à la limite admise quantifiée en terme de légumes (resp. viande).

Il en ressort que `Dlimite_viande` et `Dlimite_légumes` sont les quadruplets :

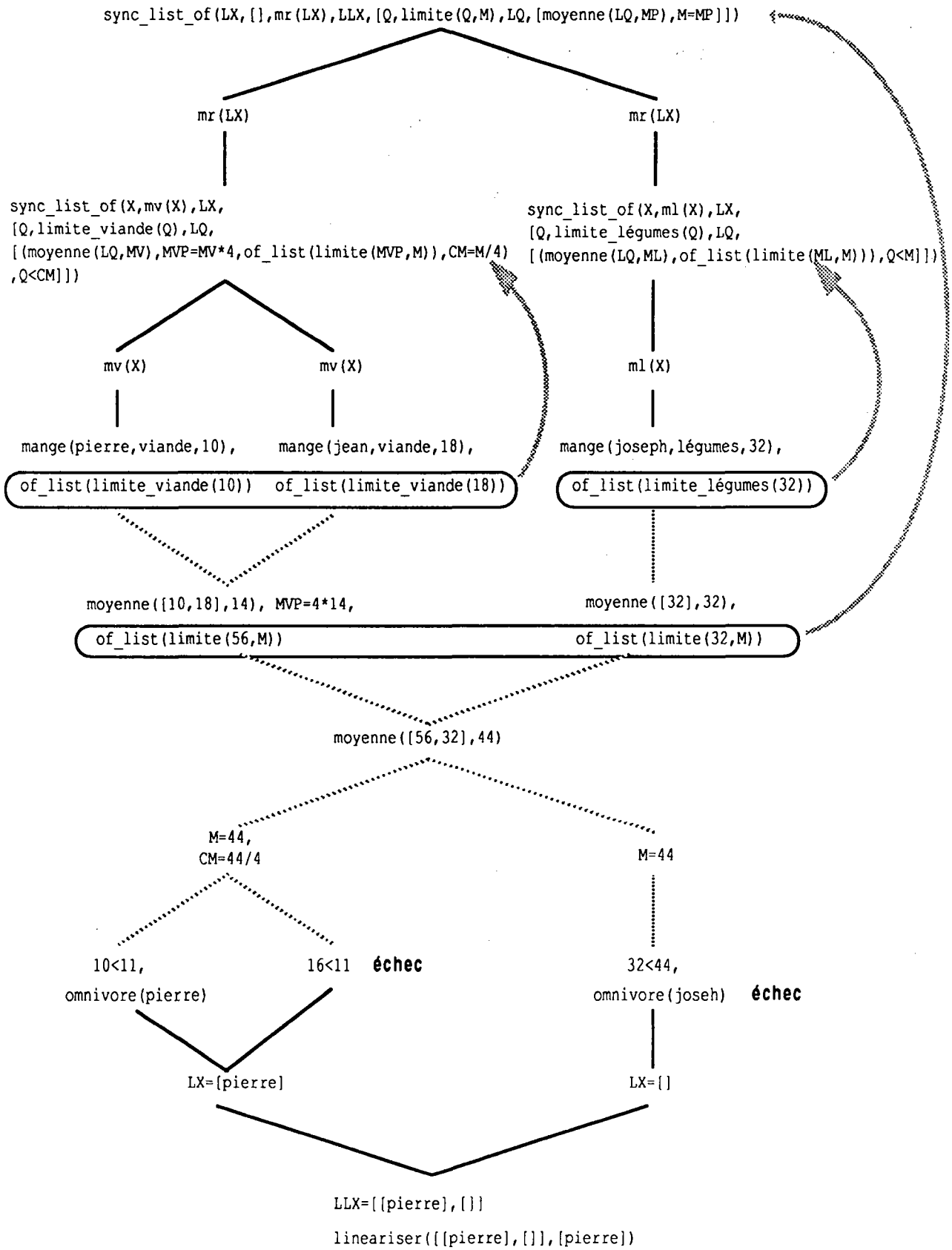
```
[Q,limite_viande(Q),LQ,[(moyenne(LQ,MV),MVP=MV*4,of_list(limite(MVP,M)),CM=M/4),
                          Q<CM
                        ]
]

[Q, limite_légumes(Q), LQ, [(moyenne(LQ,ML), of_list(limite(ML,M))),
                             Q<M
                           ]
]
```

où `of_list(limite(MVP,M))` (resp. `of_list(limite(ML,M))`) est satisfait si `MVP` (resp. `MV`) étant la moyenne des quantités mangées par les mangeurs de viande (resp. légumes) en terme de légumes, `M` est la limite admise en terme de légumes. Le quadruplet `Dlimite` s'en déduit :

```
[Q, limite(Q,M), LQ, [moyenne(LQ,MP), M = MP]]
```

La figure ci-dessous donne une représentation graphique de l'exécution du programme. Les littéraux entourés font partie d'une même collectivité. Les flèches grises désignent les littéraux où se trouvent les définitions des prédicats collectifs associés à ces collectivités. Les lignes continues renvoient aux évaluations indépendantes des littéraux dont elles sont issues (branches OU) tandis que les lignes pointillées représentent l'évaluation des prédicats collectifs. `mr`, `mv`, `ml` sont les abréviations respectives de `mangeurs_r`, `mangeur_r_viande` et `mangeur_r_légumes`.


sync_list_of(LX,[],mr(LX),LLX,[Q,limite(Q,M),LQ,[moyenne(LQ,MP),M=MP]])
mr(LX)
mr(LX)
sync_list_of(X,mv(X),LX,
[Q,limite_viande(Q),LQ,
[(moyenne(LQ,MV),MVP=MV*4,of_list(limite(MVP,M)),CM=M/4)
,Q<CM]])
sync_list_of(X,ml(X),LX,
[Q,limite_légumes(Q),LQ,
[(moyenne(LQ,ML),of_list(limite(ML,M))),Q<M]])
mv(X)
mv(X)
ml(X)
mange(pierre,viande,10),
mange(jean,viande,18),
mange(joseph,légumes,32),
of_list(limite_viande(10))
of_list(limite_viande(18))
of_list(limite_légumes(32))
moyenne([10,18],14), MVP=4*14,
moyenne([32],32),
of_list(limite(56,M))
of_list(limite(32,M))
moyenne([56,32],44)
M=44,
CM=44/4
M=44
10<11,
16<11
échec
32<44,
omnivore(pierre)
omnivore(joseh)
échec
LX=[pierre]
LX=[]
LLX=[[pierre],[]]
lineariser([[pierre],[]],[pierre])

2.5 Utilisation des variables globales

L'élégance du programme peut encore être améliorée par l'utilisation de variables globales. Au lieu de définir le prédicat `mangeurs_r` au moyen de deux clauses, une pour chaque type de nourriture, on utilise un prédicat `sync_list_of` non-déterministe suivant la nature de la nourriture ingérée. D'un point de vue opérationnel, le comportement est identique au précédent : les moyennes relatives à chaque type de nourriture sont effectuées par le prédicat collectif associé au `sync_list_of` apparaissant dans la définition du prédicat `mangeurs_r`, tandis que la moyenne générale est déterminée par le prédicat collectif associé au `sync_list_of` englobant, c'est-à-dire celui apparaissant dans la définition du prédicat `mangeurs_raisonnables`.

Voici la nouvelle définition des prédicats `mangeurs_raisonnables` et `mangeurs_r` :

```
mangeurs_raisonnables(L) :-
        sync_list_of(LX,[M],mangeurs_r(LX,M),LLX,
                        [Q,limite(Q),LQ,[moyenne(LQ,M),true]]),
        lineariser(LLX,L).

mangeurs_r(LX,M) :-
        sync_list_of(X,[B,M],(mange(X,B,Q),of_list(limite(B,Q)),omnivore(X)),LX,
                        [E,limite(C,E),LE,[moyenne_ponderee(C,LE,M,CM),E<CM]]).

moyenne_ponderee(viande,LE,M,CM) :-
        moyenne(LE,MC),
        MVP is MC*4,
        of_list(limite(MVP,M)),
        CM is M/4.
moyenne_ponderee(légumes,LE,M,M) :-
        moyenne(LE,MV),
        of_list(limite(MV,M)).
```

Notons combien il est plus aisé d'étendre ce programme à des nourriture ayant des valeurs protéiniques différentes. Il suffit pour ce faire de compléter la définition du prédicat `moyenne_ponderee`.

3 Spécification d'un problème de recherche de solutions

Nous abordons maintenant un exercice classique dont l'expression fait intervenir des prédicats collectifs "créés dynamiquement" et définis à partir de ceux "précédemment créés". Nous procédons par étape à partir d'une spécification initiale.

Le problème est formulé de la façon suivante :

- soit un jeu dont la configuration initiale est `Configinit`. Résoudre le problème consiste à déterminer les solutions telles que le chemin menant de `Configinit` à ces solutions soit de coût minimal.

- une configuration `Conf` est soit

  • une solution : `solution(Conf)` est alors satisfait

  • un nœud non solution : `non_solution(Conf)` est alors satisfait. Sachant que `Cout` est le coût d'une dérivation menant de `Configinit` à `Conf`, `descendant([Conf,Cout],[Conf1,Cout1])` est satisfait si la configuration `Conf1` peut se dériver immédiatement de `Conf` (par application de l'une des règles du jeu) et si `Cout1` est le coût de `Conf1` (avec `Cout` ≤ `Cout1`).

Une première formulation du problème est donnée par la définition du prédicat `solutions_cout_minimal(Configinit,Ls)` satisfait si `Ls` est l'ensemble des solutions de coût minimal et dérivables à partir de `Configinit`.

Voici une définition de ce prédicat.

```
solutions_cout_minimal(Configinit,Ls) :-
        list_of(R,[Configinit],recherche([Configinit,0],R),LR),
        meilleures_solutions(LR,Ls).
```

où

- `recherche([Conf,Cout],[S,C])` est satisfait si `S` est une solution de coût `C` dérivable à partir d'une configuration `Conf` de coût `Cout`

- `meilleures_solutions(LR,Ls)` est satisfait si `Ls` est la liste des solutions qui appartiennent à `LR` et qui sont de coût minimal.

Le prédicat `recherche` est défini simplement par :

```
recherche([Conf,Cout],[Conf,Cout]) :-
        solution(Conf).
recherche([Conf,Cout],R) :-
        non_solution(Conf),
        descendant([Conf,Cout],[Conf1,Cout1]),
        recherche([Conf1,Cout1],R).
```

4 Détermination des meilleures solutions en utilisant A*

On fait maintenant l'hypothèse que l'algorithme de contrôle A* [Nilsson 79] peut s'appliquer, c'est-à-dire que les conditions suivantes sont respectées :

- il existe au moins une solution,
- pour toute configuration `Conf`, il existe une fonction heuristique permettant d'estimer la valeur du coût `C` d'une dérivation menant de `Conf` à une solution minimale. Si `H` est l'estimation d'un coût `C` alors `0≤H≤C` (si `Conf` est une solution, `H=0`).

Appelons "estimé" d'un nœud `Conf` la valeur `Cout+H` : il s'agit de l'estimation du coût d'un chemin allant de la configuration initiale `Configinit` à une solution en passant par ce nœud. A* se déroule de la manière suivante : à un pas donné, l'estimé de chacun des nœuds non encore *développés* (c'est-à-dire dont on n'a pas encore calculé les configurations immédiatement dérivables) est calculé et seuls ceux d'entre eux ayant l'estimé minimal sont développés. Les conditions d'application de A* assurent qu'une solution choisie pour être développée est minimale.

La nouvelle définition de `solutions_cout_minimal` s'établit à partir de la précédente. A chaque nœud `Conf` est associé un couple `[Cout_solution,Estime]` : `Cout_solution` représente le coût de ce nœud s'il est solution et sinon vaut $+\infty$, `Estime` est l'estimé du nœud. Deux cas sont à considérer :

- `Conf` est une solution de coût `Cout`. Soient `Mincsol` le minimum des coûts des solutions déjà obtenues et `Minest` le minimum de tous les estimés des nœuds non développés (solutions comprises). `Conf` est une solution minimale si son coût `Cout` est égal à `Minest` (et dans ces conditions `Mincsol=Minest`). Sinon (si `Cout≠Minest`) cette configuration peut éventuellement être solution s'il n'existe pas, à ce pas, de solution minimale (c'est-à-dire si `Mincsol=Minest`).

- `Conf` est un nœud non solution. Soit le prédicat `heuristic(Conf,H)` satisfait si `H` est l'heuristique associée à la configuration `Conf`. Si l'estimé `Estime=Cout+H` de ce nœud est minimal (c'est-à-dire si `Estime=Minest`) le nœud `Conf` doit être développé. Si `Estime≠Minest`, il peut être développé ultérieurement dans le cas où aucune solution minimale n'existe (c'est-à-dire si `Minest≠Mincsol`).

Soit `min_csol_est` un prédicat collectif tel que `of_list(min_csol_est([Cout_solution,Estime],[Mincsol,Minest]))` est satisfait pour une configuration `Conf` à laquelle est associée `[Cout_solution,Estime]` si `Mincsol` et `Minest` sont les minimums satisfaisant les définitions précédentes. Soit également, `recherche1([Conf,Cout],S)` satisfait si `S` est une solution minimale dérivable à partir de `Conf` ayant pour coût `Cout`.

On peut définir le prédicat `solutions_cout_minimal` comme suit :

```
solutions_cout_minimal(Configinit,Ls) :-
      sync_list_of(S,[Configinit],recherche1([Configinit,0],S),Ls,
            [Ce,min_csol_est(Ce,MCe),LCe,[mg(LCe,MCE),MCE=MCe]]).
```

où `mg([[C1,E1], ...,[Cn,En]],[MC,ME])` (pour minimums globaux) est satisfait si `MC` (resp. `ME`) est le minimum de `C1, ..., Cn` (resp. `E1, ..., En`).

La définition de `recherche1([Conf,Cost],S)` se dérive facilement de la définition de `recherche` en prenant en compte les deux cas précédents :

```
recherche1([Conf,Cout],Conf) :-                    % Conf est une solution
      solution(Conf),
      meilleur_Cout(Cout).

meilleur_Cout(Cout) :-                             % solution minimale
      of_list(min_csol_est([Cout,Cout],[Cout,Cout])).
meilleur_Cout(Cout) :-                             % solution éventuellement minimale
      of_list(min_csol_est([Cout,Cout],[Mincsol,Minest])),
      Cout\==Minest,
      Mincsol\==Minest,
      meilleur_Cout(Cout).


recherche1([Conf,Cout],S) :-                       % Conf n'est pas une solution
      non_solution(Conf),
      heuristic(Conf,H),
      Estime is Cout+H,
      meilleur_descendant([Conf,Cout],Estime,S).

meilleur_descendant(N,Estime,S):-                  % noeud développé
      of_list(min_csol_est([+∞,Estime],[Mincsol,Estime])),
      descendant(N,N1),
      recherche1(N1,S).
meilleur_descendant(N,Estime,S) :-                 % noeud éventuellement développable
      of_list(min_csol_est([+∞,Estime],[Mincsol,Minest])),
      Estime\==Minest,
      Mincsol\==Minest,
      meilleur_descendant(N,Estime,S).
```

Exemple : Soit la représentation graphique suivante d'une évaluation particulière de `solutions_cout_minimal` :

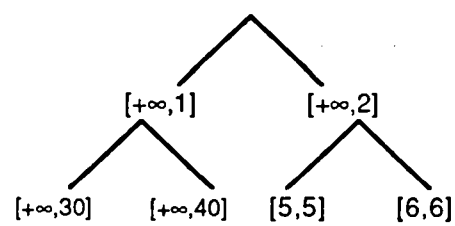


fig.1

où un couple [Cout_solution,Estime] est associé à chaque nœud : ainsi le nœud étiqueté [5,5] est un nœud terminal qui est une solution de coût 5 et le nœud [+∞,1] est un nœud non solution ayant pour estimé 1.

Les étapes successives de la recherche peuvent être figurées par le schéma plus détaillé suivant où les évaluations de min_csol_est sont représentées par des lignes pointillées et celles de recherchel par des lignes pleines : les lignes pleines verticales représentent des évaluations de meilleur_descendant dans le cas où le nœud est éventuellement développable.

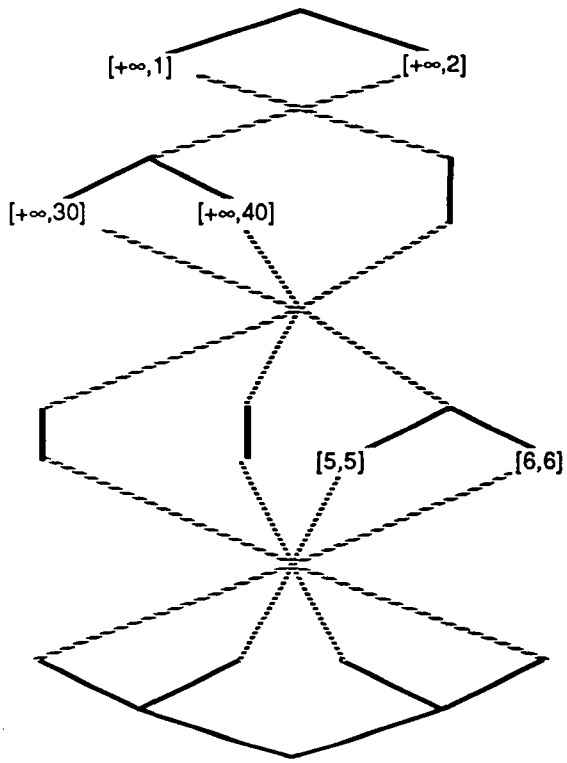


fig.2

Cette figure doit être comprise ainsi : à partir de Configinit il y a production de deux nœuds non solutions ayant respectivement pour estimé 1 et 2. Mincsol (ici +∞) et Minest (ici 1, c'est-à-dire le minimum de 1, 2) sont ensuite déterminés à l'aide du prédicat collectif. Le nœud d'estimé minimal 1 est alors développé en deux nœuds non solutions ayant respectivement pour estimé 30 et 40. Mincsol (ici +∞) et Minest (ici 2, c'est-à-dire le minimum de 30, 40, 2) sont ensuite déterminés ce qui conduit au développement du nœud d'estimé 2 en deux nœuds solutions de coût 5 et 6. Mincsol (ici 5, minimum de +∞, +∞, 5, 6) et Minest (ici également 5, minimum de 30, 40, 5, 6) sont alors déterminés et la liste de solutions LS est construite. LS est constituée ici de la seule solution de coût 5.

5 Optimisation en vue d'évaluations plus locales

Comme le montre la fig.2, l'évaluation des minimums est effectuée globalement à partir de couples `[Cout_solution,Estime]` associés aux nœuds et solutions candidats au développement. Cette évaluation pourrait être réalisée plus localement dans la mesure où min(x,y) est associatif. L'idée consiste à associer à chaque nœud susceptible d'avoir des descendants son propre prédicat collectif `min_csol_est` (dans le cas précédent, seule la racine `Configinit` en posséde un). Un tel prédicat permet de définir les "minimums locaux" (minimums du coût des solutions et minimums des estimés) concernant les descendants éventuellement développables d'un nœud : les minimums globaux attendus sont ceux associés à `Configinit`.

Pour obtenir un tel comportement, il suffit d'effectuer deux légères transformations :

(i) introduire `recherche2([Conf,Cout],Ls)` où `Ls` est l'ensemble des solutions minimales dérivables à partir de `Conf`

(ii) modéliser le développement d'un nœud `N` par un littéral de prédicat `sync_list_of` portant sur `recherche2(N1,LS)` où `N1` est un descendant de `N` et définissant un prédicat collectif `min_csol_est` tel que `of_list(min_csol_est([Cout_solution,Estime],[Mincsol,Minest]))` est satisfait si, `[Cout_solution,Estime]` étant associé à un descendant éventuellement développable de `N1`. `Mincsol` et `Minest` sont les minimums locaux concernant tous les descendants éventuellement développables de `N1`.

Le programme devient :

```
solutions_cout_minimal(Configinit,Ls) :-
        sync_list_of(LS,[Configinit],recherche2([Configinit,0],LS),LLs,
                     [Ce,min_csol_est(Ce,MCe),LCe,[mg(LCe,MCE),MCE=MCe]]),
        lineariser(LLs,Ls).
```

où `lineariser(LLs,Ls)` est satisfait si `Ls` est la liste des solutions apparaissant dans la liste de listes `LLs`.

```
recherche2([Conf,Cout],[Conf]) :-                % Conf est une solution
        solution(Conf), meilleur_Cout(Cout).

meilleur_Cout(Cout) :-
        of_list(min_csol_est([Cout,Cout],[Cout,Cout])).
meilleur_Cout(Cout) :-
        of_list(min_csol_est([Cout,Cout],[Mincsol,Minest])),
        Cout\==Minest,
        Mincsol\==Minest,
        meilleur_Cout(Cout).

recherche2([Conf,Cout],Ls) :-                    % Conf n'est pas une solution
        non_solution(Conf),
        heuristic(Conf,H),
        Estime is Cout+H,
        meilleur_descendant([Conf,Cout],Estime,Ls).
```

```
meilleur_descendant(N,Estime,Ls):-
        of_list(min_csol_est([+∞,Estime],[Mincsol,Estime])),
        descendant(N,N1),
        sync_list_of(LS,[N1],recherche2(N1,LS),LLs,
                        [Ce,min_csol_est(Ce,MCe),LCe,[mmp(LCe,MCE),MCE=MCe]]).
        lineariser(LLs,Ls).
meilleur_descendant([Conf,Cout],Estime,Ls):-
        of_list(min_csol_est([+∞,Estime],[Mincsol,Minest])),
        Estime\==Minest,
        Mincsol\==Minest,
        meilleur_descendant([Conf,Cout],Estime,Ls).
```

mmp (pour minimums des minimums partiels) est défini par :

```
mmp(LCe,MCe) :- mg(LCe,PMCe), of_list(min_csol_est(PMCe,MCe)).
```

Exemple : la représentation graphique correspondant à l'évaluation de l'exemple précédent devient :

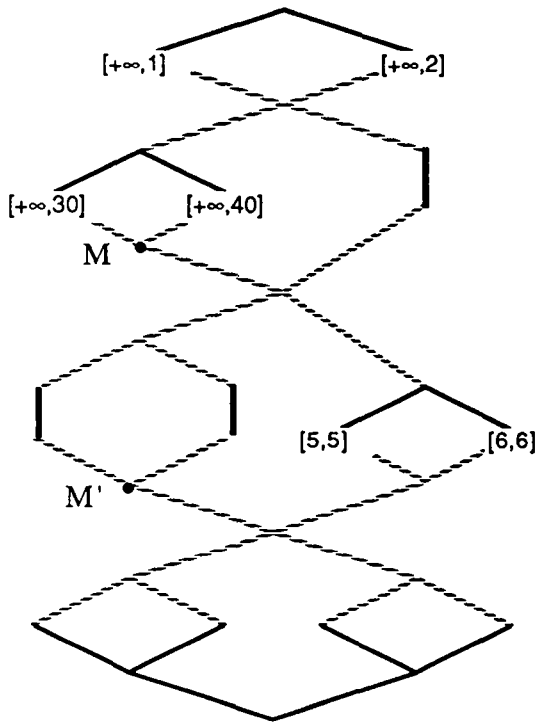


fig.3

6 Elimination des évaluations redondantes

Un simple coup d'œil à la fig.3 permet de constater la présence d'évaluations redondantes. Le point M représente une étape dans laquelle les minimums de `[+∞,+∞]` et `[30,40]` sont calculés : le même calcul se reproduit en M'. Pour pallier ce défaut on considère la "pertinence" des minimums globaux `Mincsol`, `Minest` vis-à-vis des descendants d'un nœud donné. Ils ne sont pertinents pour ces descendants que dans deux cas (non exclusifs) : (i) un successeur (descendant immédiat ou plus

lointain) de ce nœud doit être développé, (ii) il existe une solution minimale.

Cette optimisation est rendue à l'aide du prédicat `recherche3([Conf,Cout],Ls)` dont la seule différence avec `recherche2` concerne le prédicat collectif attaché à chaque nœud développé. Cette fois, `of_list(min_csol_est([Cout_solution,Estime],[Mincsol,Minest]))` est satisfait si les conditions précédentes sont remplies et si, de plus, `Mincsol` et `Minest` sont pertinents pour le nœud auquel est associé `[Cout_solution,Estime]`. Cela se traduit par une définition différente de mmp soit :

```
mmp(LCe,MCe) :- mg(LCe,PMCe), di(PMCe,MCe).
```

Les clauses définissant `di` (pour descendant intéressé) sont les suivantes :

```
di([PMC,PME],[MC,PME]) :-
       of_list(min_csol_est([PMC,PME],[MC,PME])).      % cas (i)
di([PMC,PME],[MC,MC]) :-
       of_list(min_csol_est([PMC,PME],[MC,MC])),       % cas (ii)
       MC\==PME.
di([PMC,PME],[MC,ME]):-
       of_list(min_csol_est([PMC,PME],[MC1,ME1])),     % descendant éventuellement
       PME\==ME1,                                      % intéressé
       ME1\==MC1,
       di([PMC,PME],[MC,ME]).
```

Exemple : la représentation graphique de la nouvelle évaluation est la suivante :

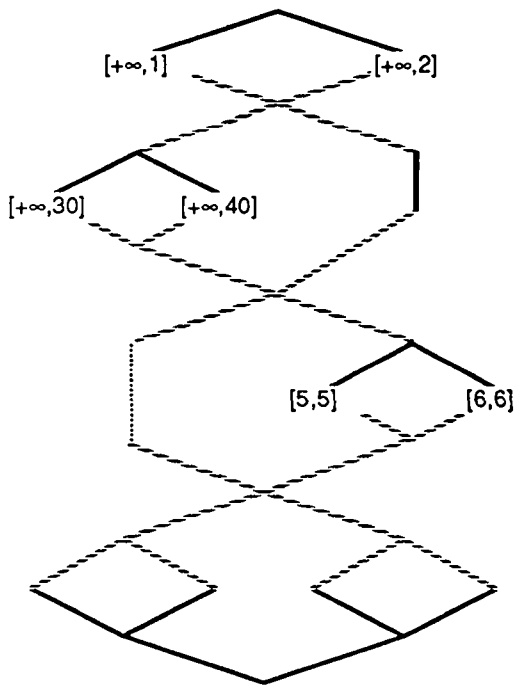


fig.4

7 Cas particuliers

Nous n'avons pas rencontré jusqu'à présent de cas où tous les littéraux `of_list(P)` d'une même collectivité ont la valeur faux du fait de l'évaluation de la liste de buts `G`. Ceci se produit si on effectue l'optimisation (très légère) qui consiste à considérer dans notre expression de A* que `Mincsol` et `Minest` ne sont pas, en fait, pertinents dans le cas (ii) précédent : en effet, s'il existe une solution minimale d'un coût inférieur aux estimés ou coûts solutions des successeurs d'un nœud, ces successeurs ne seront jamais développés et ne seront donc pas solution minimale. Cette optimisation se traduit par la suppression de la deuxième règle définissant le prédicat `di`.

La représentation graphique de cette évaluation particulière devient alors :

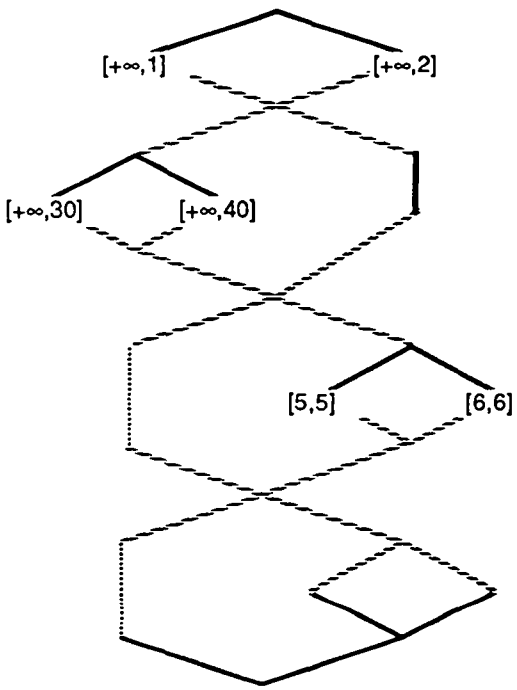


fig.5

8 Extensions possibles

Une première extension permet d'associer plusieurs prédicats collectifs à `sync_list_of`.

Une seconde extension consiste à introduire une notion analogue pour les branches ET : soient `sync_par_and([F1,...,Fn],[Y,P,LY,G])` satisfait si `F1` ∧ ... ∧ `Fn` est satisfait et `and_par(P0)` (supposé présent dans les résolvantes associées à `F1,...,Fn`) défini de la même manière que `of_list(P0)` (à la différence près que, cette fois, les résolvantes associées à un `sync_par_and` sont

celles produites par F1,...,Fn).

Par exemple, le prédicat meme_mot_des_feuilles(X,Y) satisfait si les arbres binaires X et Y possèdent le même mot des feuilles, serait défini de la manière suivante au moyen de ces prédicats :

```
meme_mot_des_feuilles(X,Y) :-
       sync_par_and([mot_des_feuilles(X),mot_des_feuilles(Y)],
                    [Y,meme_feuilles(Y),[R,R],true]).

mot_des_feuilles(X) :-
       feuille(X),
       and_par(meme_feuilles(X)).
mot_des_feuilles([U,V]) :-
       mot_des_feuilles(U),
       mot_des_feuilles(V).
```

9 Implantation

Au premier abord, il nous avait semblé qu'une implantation directe était possible grâce au prédicat freeze de Prolog II. Cependant, freeze est inopérant "à l'intérieur" d'un bag_of implanté de la manière habituelle. Il ne peut donc être utilisé que pour l'implantation de sync_par_and et and_par. Pour simuler sync_list_of et of_list, nous avons dû écrire un interprète Prolog ayant une stratégie particulière (proche de largeur d'abord) et une implantation particulière de bagof. L'annexe 3 contient une présentation plus détaillée de l'implantation, en particulier le programme de l'interprète.

Il serait certainement instructif d'étudier l'implantation de ces prédicats dans un environnement parallèle [Syre 85, Takeuchi 86], où of_list(P0) (resp. and_par(P0)) seraient considérés comme des littéraux dont l'évaluation est suspendue jusqu'à la construction complète de la liste LE.

On peut imaginer, enfin, une évaluation de of_list(P0) (resp. and_par(P0)) cadencée par les données. Elle pourrait être réalisée avant la construction complète de la liste correspondante LE, ce qui conduirait en cas de succès (resp. d'échec) à relancer immédiatement l'évaluation de la résolvante contenant ce prédicat of_list(P0) (resp. à stopper immédiatement l'évaluation de toutes les résolvantes liées à ce sync_list_of).

Remerciements : Nous tenons à remercier J.P.Banatre qui a introduit, l'idée sous-jacente à la notion de prédicat collectif dans sa thèse [Banatre 81].

Références

[Banatre 81] J.P.Banatre, Contribution à l'étude et d'outils de construction de programmes parallèles et fiables
Thèse d'état. Université de Rennes. 1981

[Colmérauer 83]A.Colmérauer, H.Kanoui, M.Van Caneghem, Prolog, bases théoriques et développements actuels
Technique et Science Informatiques, vol. 2, 4, 1983

[Cox 85] P.T.Cox, T.Pietrzykowski, Surface deduction: a uniform mechanism for logic programming
2nd Symposium on Logic Programming, Boston, 1985

[Giannesini 85] F.Giannesini, H.Kanoui, R.Pasero, M.Van Caneghem, Prolog
InterEditions, 1985

[Nakamura 86]K.Nakamura, Heuristic Prolog : logic program execution by heuristic search
Third Logic Programming Conference, London, 1986

[Nilsson 79] N.Nilsson, Principles of Artificial Intelligence
Tioga, 1979

[Naish 84] L.Naish, All Solutions Predicates in PROLOG
TR 84/4, University of Melbourne, 1984

[Porto 82] A.Porto, Epilog a language for extended programming in logic
First Logic Programming Conference, Marseille, 1982

[Quiniou 85] R.Quiniou, L.Trilling, Des primitives pour la synchronisation de branches OU. Un exemple d'application
Journées Programmation en Logique, Trégastel, France, 1985

[Quintus 87] Quintus Prolog, User's Guide and Reference Manual, 1987

[Syre 85] J.C.Syre, Une revue de modèles parallèles pour Prolog
Journées Programmation en Logique, Trégastel, France, 1985

[Takeuchi 86] A.Takeuchi, K.Furukawa, Parallel logic programming languages
ICOT, TR-163, 1986

[Warren 82] D.H.D. Warren, High Order Extension to PROLOG : Are They Needed?
Machine Intelligence 10, 1982

ANNEXE 1 : Définition de `sync_list_of` et de `of_list` en fonction de `bagof`.

On considère d'abord le cas simple suivant :

(1)
```
sync_list_of(X,LVG,F,LX,
             [E,PC,LE,[G,L]])
```

où `F` est une formule dont les démonstrations ne font pas intervenir de littéraux de la forme `of_list(PC',R)` .

La formule (1) est alors équivalente à :

(1')
```
bagof(X,LVG,F,LX)
```

L'autre cas possible est donné par :

(2)
```
sync_list_of(X,LVG,(F,of_list(PC',R)),LX,
             [E,PC,LE,[G,L]])
```

où `F` satisfait aux mêmes conditions que précédemment et où les démonstrations de `R` peuvent éventuellement mener à l'évaluation de littéraux de prédicat `of_list`.

Soit `LVL` l'ensemble des variables de `F`, `PC'`, `R` n'appartenant pas à `LVG` : ces variables sont locales.

Soit `LVG'` l'ensemble des variables de `LE` et de `G` n'appartenant pas à `LVG` : ces variables sont globales (au même titre que celles appartenant à `LVG`). On impose `LVL` $\cap$ `LVG'` $= \emptyset$ de façon à ce que toutes les variables présentes dans `LE` et `G` soient globales.

Soit `LVL'` l'ensemble des variables de `E`, `PC`, `L` n'appartenant pas aux autres ensembles considérés, c'est-à-dire n'appartenant pas à `LVG` $\cup$ `LVL` $\cup$ `LVG'` : ces variables sont locales.

Soit `TVL` un n-uplet ordonné (par exemple, lexicographiquement) des variables locales, c'est-à-dire celles appartenant à `LVL` $\cup$ `LVL'`.

La formule (2) est équivalente à :

```
bagof([TVL,E],LVG,(F, PC=PC'),LTVLE),
projection-1(LTVLE,LTVL),
projection-2(LTVLE,LE),
G,
sync_list_of(X,LVG ∪ LVG' ∪{LTVL},
             (TVL ∈ LTVL, L, R),
             LX,
             [E*,PC*,LE*,[G*,L*]] )
```

où:

`projection-1(LTVLE,LTVL)` (resp. `projection-2(LTVLE,LE)`) est satisfait si `LTVL` (resp. `LE`) est la liste constituée des premiers (resp. seconds) éléments des couples constituant `LTVLE`.

`E*,PC*,LE*,G*,L*` sont des variantes des `E, PC, LE, G, L` dans lesquelles les variables appartenant à `LVG' ∪ LVL'` sont remplacées par de nouvelles variables.

Exemple :

```
sync_list_of(X,[],(mangeurs(X,B,Q),of_list(limite(Q),omnivore(X))),LX,
             [E,limite(E),LE,[moyenne(LE,M),E<M]])
```

est équivalent à :

```
bagof([[B,E,Q,X],E],[],(mangeurs(X,B,Q),limite(E)=limite(Q)),LTVLE),
projection-1(LTVLE,LTVL),
projection-2(LTVLE,LE),
moyenne(LE,M),
sync_list_of(X,[LE,M,LTVL],([B,E,Q,X] ∈ LTVL, E<M, omnivore(X)), LX,
             [E*,limite(E*),LE*,[moyenne(LE*,M*),E*<M*]])
```

On a ici :

```
LVG = []
LVL = [X,B,Q]
LVG' = [LE,M]
LVL' = [E]
TVL = [B,E,Q,X]
```

et, en tenant compte de la définition de `mange` et de `omnivore` :

```
LTVL = [[pierre,10,10,viande],[jean,18,18,viande],[joseph,32,32,légumes]]
LE = [10,18,32]
```

ANNEXE 2 : Définition récursive de `bagof` et de `sync_list_of`

On donne ici les règles permettant d'inférer `bagof(X,LVG,F,LX)` et dans un second stade celles permettant d'inférer `sync_list_of(X,LVG,F,LX,[E,PC,LE,[G,L]])`. Ces dernières respectent l'Annexe 1. Nous avons repris les notations utilisées dans [Cox 85] où les substitutions opérées au cours d'une résolution apparaissent directement dans les résolvantes.

On trouve dans [Naish 84] une étude détaillée sur l'implantation des prédicats du type de `bagof`. Il en ressort que i) leur définition doit clairement distinguer les variables locales des variables globales, ii) que la liste des solutions `LX` doit être fournie sous une forme canonique (c'est-à-dire avoir ses éléments triés selon un certain ordre), iii) les éléments de cette liste ne doivent pas comporter de variables locales, iiii) enfin, pour pouvoir distinguer les ensembles de solutions associés à des valeurs de variables globales il est nécessaire de disposer d'un prédicat `noneg(X,Y)` correctement implanté.

Ici, nous n'avons pas pris en compte le point ii) dans la mesure où nous ne considérons pas comme fondamental l'ordre des éléments de `LX` et le point iii) en admettant qu'il s'agit là d'un aspect à laisser à l'appréciation du programmeur ([Naish 84] adopte la même position).

Les prédicats `bagof` et `sync_list_of` sont définis par :

```
bagof(X,LVG,F,LX) ← bagofp(X,LVG,F,LXLG),
                    selection(LXLG,LG,LX)

sync_list_of(X,LVG,F,LX,Q) ← sync_list_ofp(X,LVG,F,LXLG,Q),
                              selection(LXLG,LG,LX)
```

Le prédicat `bagof(X,LVG,F,LX)` (resp. `sync_list_of(X,LVG,F,LX,Q)`) est satisfait si `LXLG` est une liste de tous les couples `(X,LG)` où `X` est une solution pour `F` (resp. pour `F` étant donné `Q`) et `LG` est la liste des valeurs des variables globales (c'est-à-dire de `LVG`) correspondantes.

Le prédicat `selection(LXLG,LG,LX)` est satisfait si `LX` est une liste exhaustive de solutions extraites de `LXLG` auxquelles correspondent les mêmes valeurs `LG` des variables globales.

Il est défini par :

```
selection([],LG,[]) ←
selection([[X,LG]|LXLG],LG,[X|LX]) ← selection(LXLG,LG,LX)
selection([[X,LGp]|LXLG],LG,LX) ← noneg(LGp,LG), selection(LXLG,LG,LX)
```

Le prédicat `bagofp(X,LVG,F,LX)` est défini à l'aide des trois règles suivantes par induction structurelle sur les valeurs de `F` :

(1) `bagofp(X,LVG,□,[])`

où □ est la clause vide

(2) $\texttt{bagofp(X,LVG,}\ (H_1 \wedge \sigma_1) \vee \ldots \vee (H_{i-1} \wedge \sigma_{i-1}) \vee \mu \vee (H_{i+1} \wedge \sigma_{i+1}) \vee \ldots \vee (H_n \wedge \sigma_n)\texttt{,[[X}\mu\texttt{, LVG}\mu\texttt{] |LXLG])} \leftarrow$

$\texttt{bagofp(X,LVG,}\ (H_1 \wedge \sigma_1) \vee \ldots \vee (H_{i-1} \wedge \sigma_{i-1}) \vee (H_{i+1} \wedge \sigma_{i+1}) \vee \ldots \vee (H_n \wedge \sigma_n)\texttt{,LXLG)}$

où H_i pour `i:1,m` sont des listes de buts et σ_i pour `i:1,m` et μ sont des substitutions.

(3) $\texttt{bagofp(X,LVG,}\ H_1 \vee \ldots \vee H_{i-1} \vee ((F,R) \wedge \sigma) \vee H_{i+1} \vee \ldots \vee H_n\texttt{, LXLG)} \leftarrow$

$\texttt{bagofp(X,LVG,}\ H_1 \vee \ldots \vee H_{i-1} \vee ((G_1,R) \wedge \theta_1) \vee \ldots \vee ((G_m,R) \wedge \theta_m) \vee H_{i+1} \vee \ldots \vee H_n\texttt{, LXLG)}$

où `F` est un littéral tel que $\texttt{F}_1$ `:-` $\texttt{G}_1$, ..., $\texttt{F}_m$ `:-` $\texttt{G}_m$ sont toutes les clauses telles que $\texttt{F}_i\nu_i = (\texttt{F}\sigma)\nu_i, \nu_i$ pgu

et $\theta_i = \nu_i \circ \sigma$ pour `i:1,m`

Le prédicat `sync_list_ofp` est défini à l'aide de trois règles déduites des trois précédentes en remplaçant `bagofp` par `sync_list_ofp` et en rajoutant un cinquième paramètre `[E,PC,LE,[G,L]]` à `bagofp` et d'une quatrième déduite de la règle (2') de l'Annexe 1 :

(4) $\texttt{sync\_list\_ofp(X,LVG,(of\_list(PC}_1\texttt{,R}_1\texttt{)} \wedge \sigma_1) \vee \ldots \vee \texttt{(of\_list(PC}_n\texttt{,R}_n\texttt{)} \wedge \sigma_n)\texttt{, LXLG,}$

$\texttt{[E,PC,LE,[G,L]])} \leftarrow$

$\texttt{bagof(E,LVG,((PC}_1\texttt{=PC} \wedge \sigma_1) \vee \ldots \vee (\texttt{PC}_n\texttt{=PC} \wedge \sigma_n))\texttt{,LE),}$

`G,`

$\texttt{sync\_list\_ofp(X,LVG} \cup \texttt{LVG',((L,R'}_1\texttt{)} \wedge \theta_1) \vee \ldots \vee (\texttt{(L,R'}_p\texttt{)} \wedge \theta_p)\texttt{, LXLG,}$

`[E*,PC*,LE*,[G*,L*]])`

avec $\texttt{R'}_1, \ldots, \texttt{R'}_p$ toutes les formules appartenant à $\{ R_1, \ldots, R_n \}$ telles que $\texttt{R'}_j = R_i$, pour `j:1,p` et `i:1,n`, ssi $(\texttt{PC}_i\sigma_i)\nu_i = \texttt{PC}\nu_i, \nu_i$ pgu et $\theta_i = \nu_i \circ \sigma_i$ pour `i:1,n`

Exemple:

```
sync_list_ofp(X,[],(mangeurs(X,B,Q) ∧ of_list(limite(Q),omnivore(X))),LX,
            [E,limite(E),LE,[moyenne(LE,M),E<M]])
```

se déduit de (en employant la règle (3)) :

```
sync_list_ofp(X,[], (of_list(limite(Q),omnivore(X))∧(X=pierre,B=viande,Q=10)) ∨
                    (of_list(limite(Q),omnivore(X))∧(X=jean,B=viande,Q=18)) ∨
                    (of_list(limite(Q),omnivore(X))∧(X=joseph,B=légumes,Q=32)),
             LX,
             [E,limite(E),LE,[moyenne(LE,M),E<M]])
```

qui se déduit de (en employant la règle (4)) :

```
bagofp(E,[],
      (limite(10)=limite(E) ∧(X=pierre,      B=viande,       Q=10)) ∨
      (limite(18)=limite(E) ∧(X=jean,        B=viande,       Q=18)) ∨
      (limite(32)=limite(E) ∧(X=joseph,      B=légumes,      Q=32)),
      LE),
moyenne(LE,M),
sync_list_ofp(X,[LE,M],
            ((E<M, omnivore(X)) ∧(X=pierre, B=viande,  Q=10,  E=10)) ∨
            ((E<M, omnivore(X)) ∧(X=jean,   B=viande,  Q=18,  E=18)) ∨
            ((E<M, omnivore(X)) ∧(X=joseph, B=légumes, Q=32,  E=32)),
            LX,
            [E*,limite(E*),LE*,[moyenne(LE*,M*),E*<M*]])
```

ANNEXE 3 : implantation en Prolog [Quintus 87]

Nous présentons ici le méta-interpréteur Prolog permettant l'interprétation des programmes comportant les primitives `sync_list_of` et `of_list`. Avant de donner le texte du programme lui-même nous donnons quatre versions de méta-interpréteur Prolog se rapprochant de plus en plus de notre propre version.

Voici tout d'abord un méta-interpréteur classique de Prolog. Nous avons fait apparaitre explicitement la règle d'interprétation d'un prédicat `bag_of` chargé de construire l'ensemble des solutions satisfaisant un but donné.

```
pas_int0(true).
pas_int0((A,B)) :- pas_int0(A), pas_int0(B).
pas_int0(A) :- system(A).

pas_int0(bag_of(X,Goal,Xs)) :- bagof(X,pas_int0(Goal),Xs).
pas_int0(A) :- clause(A,B), pas_int(B).
```

Si l'on veut permettre un traitement de la résolvante (réordonnancer les sous-buts, par exemple) il faut faire apparaitre explicitement la construction de celle-ci . Un pas d'interprétation (`pas_int1`) est une suite de pas de résolution (`pas_res1`) qui transforme la résolvante de façon atomique (réécriture d'un littéral par pas de résolution).

```
pas_int1(true).
pas_int1((B,LB)) :- pas_res1(B,LB,R), pas_int1(R).
pas_int1(B) :- pas_res1(B,true,R), pas_int1(R).

pas_res1(bag_of(X,Goal,Xs),LB,LB) :- bagof(X,pas_int1(Goal),Xs).
pas_res1(B,LB,LB) :- system(B), call(B).
pas_res1(B,LB,R) :- clause(B,LB1), conc(LB1,LB,R).
```

Caractéristique essentielle de l'interprétation de nos primitives, l'évaluation d'une résolvante peut être interrompue. On rajoute au prédicat assurant l'interprétation un argument représentant la partie de résolvante non évaluée avant suspension. Dans ce cas `pas_int2(LB,R)` est satisfait si `R` est la résolvante demeurant non évaluée après l'interprétation de la liste de buts `LB`.

```
pas_int2(true,true).
pas_int2((B,LB),R) :- pas_res2(B,LB,Ri), pas_int2(Ri,R).
pas_int2(B,R) :- pas_res2(B,true,Ri), pas_int2(Ri,R).

pas_res2(bag_of(X,Goal,Xs),LB,LB) :- bagof(X,pas_int2(Goal,true),Xs).
pas_res2(B,LB,LB) :- system(B), call(B).
pas_res2(B,LB,R) :- clause(B,LB1), conc(LB1,LB,R).
```

La suspension de résolvante se produit au cours de l'évaluation de `sync_list_of` qui est une extension de `bag_of`. Ces résolvantes suspendues sont réactivées après évaluation d'un prédicat collectif. Une gestion de la liste des résolvantes est donc nécessaire. Celle-ci est assurée au cours de

l'interprétation de bag_of par le prédicat list_pas_int3.

```
pas_int3(true,true).
pas_int3((B,LB),R) :- pas_res3(B,LB,R0), pas_int3(R0,R).
pas_int3(B,R) :- pas_res3(B,true,R0), pas_int3(R0,R).

pas_res3(bag_of(X,Goal,Xs),LB,LB) :- list_pas_int3([[X,Goal]],Xs).
pas_res3(B,LB,LB) :- system(B), call(B).
pas_res3(B,LB,R) :- clause(B,LB1), conc(LB1,LB,R).

list_pas_int3([],[]).
list_pas_int3([[X,LB]|LLB],Xs) :-
       bagof([X,R],pas_int3(LB,R),LXR),
       append(LXR,LLB,LR),
       list_pas_int3(LR,Xs).
```

Le programme Prolog ci-dessous est une simple extension du méta-interpréteur précédent permettant la mise en œuvre les primitives sync_list_of et of_list. Le parallélisme OU y est bien entendu simulé mais en suivant la définition de l'annexe 2. La méthode utilisée est simple : toutes les résolvantes possibles sont construites et maintenues en même temps en mémoire. Chaque fois que plusieurs têtes de clauses peuvent s'unifier avec le littéral se trouvant en tête de résolvante, autant de résolvantes sont créées, constituées du corps de la clause et d'une copie de l'ancienne résolvante privée du littéral résolu (pourvues des substitutions convenables). L'évaluation d'une résolvante s'arrête en cas de succès (résolvante vide) ou d'échec et est suspendue si un littéral of_list s'y trouve en tête. Dans ce dernier cas l'évaluation de la résolvante suspendue ne reprend qu'après que les évaluations de toutes les autres résolvantes soient arrêtées ou suspendues. La stratégie est donc un mélange de "profondeur d'abord" (jusqu'à rencontre d'un littéral of_list) et de "largeur d'abord" (gestion de l'ensemble des résolvantes). Lorsqu'aucune résolvante ne peut être évaluée, le prédicat collectif correspondant aux résolvantes suspendues est évalué. Il faut noter que cette évaluation peut à son tour faire intervenir un prédicat collectif.

Plus précisément :

pas_int(LB,R) où LB est une liste de buts (représentée sous la forme $(B_1, \ldots, B_n)$) est satisfait si R est la résolvante obtenue après un "pas d'interprétation". Un tel pas se caractérise par l'obtention soit d'une résolvante vide en cas de succès (représentée par true) soit d'une résolvante réduite à un littéral of_list(PC,R) (représenté ici sous la forme of_listp(PCP,R) dans la mesure où nous avons réservé l'identificateur of_list au prédicat étendu introduit en 2.5).

pas_res(B,LB,R) où B est un but et LB une liste de buts est satisfait si R est la résolvante obtenue à partir de (B,LB) après application sur B d'un pas de résolution. Le nouveau cas à considérer par rapport à un interpréteur classique de Prolog est celui où B est un littéral de la forme sync_list_of(X,LVG,F,LX,[E,PC,LE,[G,L]]) : la résolvante R est alors construite à partir de la

liste de buts G et du littéral sync_list_of obtenus par application de la règle (4) de l'Annexe 2 et de LB (pourvus des substitutions convenables). C'est ce qu'on a appelé pas de résolution du prédicat collectif (cf. pas_res_pc dans le programme).

Par exemple, le but suivant est satisfait :

```
pas_res(sync_list_of(X,[],(p(U),of_listp(pc(U,V),q(L,X))),LX,
                        [E,pc(E,L),LE,[g(LE),r(X,E,L)]]),
        LB,
        (g([1,2]),
          slo([],
              [[[L1],X1,(r(X1,1,L1),q(L1,X1))],[[L2],X2,(r(X2,2,L2),q(L2,X2))]],
              LX,
              [[],[L],E,pc(E,L),LE,[g(LE),r(X,E,L)]]
        ))
```

p étant défini par les clauses :

```
        p(1).
        p(2).
```

La représentation adoptée pour un littéral sync_list_of est la suivante :

```
slo(LVG,[[LVL1,X1,H1], ..., [LVLn,Xn,Hn]],[LVG,LVL,E,PC,LE,[G,L]],LX)
```

Exemple :

```
pas_res( sync_list_of(X,[],(p(U),of_listp(pc(U,V),q(L,X))),LX,
                        [E,pc(E,L),LE,[g(LE),r(X,E,L)]]),
         LB,
         R)
```

est impliqué par :

```
pas_res(slo([],[[[U],X,(p(U),of_listp(pc(U,V),q(L,X)))]],
               [[],[L],E,pc(E,L),LE,[g(LE),r(X,E,L)]],LX),
            LB,R)
```

et qui est satisfait si R est la résolvante :

```
(g([1,2]),
 slo([],[[[L1],X1,(r(X1,1,L1),q(1,X1))],[[L2],X2,(r(X2,2,L2),q(2,X2))]],
     [[],[L],E,pc(E,L),LE,[g(LE),r(X,E,L)]],
         ),
 LB)
```

liste_pas_int(LVG,SXF,SXF1,LX1) où SXF et SXF1 sont des listes de la forme [[LVL1,X1,H1], ..., [LVLn,Xn,Hn]] et LX1 une liste de termes est satisfait si SFX1 est la liste des résolvantes obtenues après un pas d'interprétation des éléments de SFX et si LX1 est la liste des Xi solutions c'est-à-dire associés à une résolvante vide (la séparation entre résolvantes suspendues et solutions est assurée par le prédicat res_sol). La définition de ce prédicat utilise un prédicat bag_of

qui est lui-même interprété selon le schéma donné dans le quatrième meta-interpréteur préliminaire présenté plus haut. Plus précisément, l'évaluation de la liste de but de `bag_of` est assurée par le méta-interpréteur et non directement par l'interpréteur Prolog lui-même.

Exemple :

Le but

```
liste_pas_int([],
  [[[L],X,(p(U),of_listp(pc(U,V),q(L,X)))]],
  [[[L1],X1,of_listp(pc(1,V1),q(L1,X1))],[[L2],X2,of_listp(pc(2,V2),q(L2,X2))]],
  [])
```

est satisfait.

`conclb(LB1,LB2,LB)` est satisfait si la liste de but `LB` est la concaténation des listes de buts `LB1` et `LB2`.

`append(L1,L2,L)` est satisfait si la liste `L` est la concaténation des listes `L1` et `L2`.

Voici le programme de l'interpréteur :

```
interpretation(R) :-
       pas_int(R,true).

pas_int(true,true) :- !.
pas_int((of_list(PCP),R),of_listp(PCP,R)) :- !.
pas_int((B,LB),R) :- !,
       pas_res(B,LB,R0),
       pas_int(R0,R).
pas_int(of_list(P),of_listp(P,true)) :- !.
pas_int(B,R) :- pas_res(B,true,R0), pas_int(R0,R).

pas_res(sync_list_of(X,LVG,F,LX,[E,PC,LE,[G,L]]),LB,R) :-
   construire_liste_variables_locales([LVG,LE,G],[F,E,PC,L],LVL),
       !, pas_res(slo(LVG,[[LVL,X,F]],[LVG,LVL,E,PC,LE,[G,L]],LX),LB,R).
pas_res(slo(LVG,LLXF,Q,LX),LB,R) :-
       !, dif(LLXF,[]),
       liste_pas_int(LVG,LLXF,LLXFp,LXp),
       pas_res_pc(LVG,LLXFp,LXp,Q,LB,LX,R).
pas_res(B,LB,LB) :- system(B), !, call(B).
pas_res(B,LB,R) :- clause(B,PD), conc_lb(PD,LB,R).

pas_res_pc(LVG,[],LX,Q,LB,LX,LB).
pas_res_pc(LVG,LLXFp,LXp,Q,LB,LX,R) :-
       unification_pc(Q,[LVG,LE,G],LLXFp,LE,LLXFr),
       conc_lb(G,(slo(LVG,LLXFr,Q,LXr),LB),R),
       append(LXp,LXr,LX).

unification_pc(Q,[LVG,LE,G],[],[],[]).
unification_pc(Q,[LVG,LE,G],[[LVL,X,of_listp(PCP,R)]|S],
                  [E|RE],[[LVL,X,(L,R)]|S1]) :-
       copy_term(Q,Q1),
       Q1=[LVG,LVL,E,PCP,LE,[G,L]],!,
       unification_pc(Q,[LVG,LE,G],S,RE,S1).
unification_pc(Q,[LVG,LE,G],[[LVL,X,of_listp(PCP,R)]|S],[E|RE],S1) :-
       copy_term(Q,Q1),
       unification_pc(Q,[LVG,LE,G],S,RE,S1).

liste_pas_int(LVG,[],[],[]).
liste_pas_int(LVG,[[LVL,X,F]|S],LLXF,LX) :-
       bag_of(LVG-[LVL,X],LVG,F,Bag),
       res_sol(LVG,Bag,LLXFp,LXp),
       append(LLXFp,LLXFr,LLXF),
       append(LXp,LXr,LX),
       liste_pas_int(LVG,S,LLXFr,LXr).

res_sol(LVG,[],[],[]).
res_sol(LVG,[[LVG-[LVL,X],true]|SGXF],R,[X|S]) :-
       res_sol(LVG,SGXF,R,S).
res_sol(LVG,[[LVG-[LVL,X],F]|SGXF],[[LVL,X,F]|R],S) :-
       dif(F,true),
       res_sol(LVG,SGXF,R,S).
```

PI 542 **A NEW APPROACH TO VISUAL SERVOING IN ROBOTICS**
Bernard ESPIAU, François CHAUMETTE, Patrick RIVES
Juillet 1990, 44 Pages.

PI 543 **SIMPLE DISTRIBUTED SOLUTIONS TO THE READERS-WRITERS PROBLEM**
Michel RAYNAL
Juillet 1990, 10 Pages.

PI 544 **IMPLEMENTATION AND EVALUATION OF DISTRIBUTED SYNCHRONIZATION ON A DISTRIBUTED MEMORY PARALLEL MACHINE**
André COUVERT, René PEDRONO, Michel RAYNAL
Juillet 1990, 14 Pages.

PI 545 **ESTIMATION OF NETWORK RELIABILITY ON A PARALLEL MACHINE BY MEANS OF A MONTE CARLO TECHNIQUE**
Mohamed EL KHADIRI, Raymond MARIE, Gerardo RUBINO
Août 1990, 20 Pages.

PI 546 **LIMIT THEOREMS FOR MIXING PROCESSES**
Bernard DELYON
Septembre 1990, 22 Pages.

PI 547 **PERFORMANCES DES COMMUNICATIONS SUR LE T-NODE**
Frédéric GUIDEC
Septembre 1990, 38 Pages.

PI 548 **LES PREDICATS COLLECTIFS : UN MOYEN D'EXPRESSION DU CONTROLE DU PARALLELISME *OU* EN PROLOG**
René QUINIOU, Laurent TRILLING
Septembre 1990, 34 Pages.

PI 549 **NORMALISATION SOUS HYPOTHESE D'ABSENCE DE LIEN APPLICATION AU CAS NOMINAL**
François DAUDE
Septembre 1990, 42 Pages.

PI 550 **MULTISCALE SIGNAL PROCESSING : FROM QMF TO WAVELETS**
Albert BENVENISTE
Septembre 1990, 28 Pages.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399